Research Project in Mechanical Engineering - 034381

Lego Spike: Balancing Segway

Name	ID	Email	Submission date
Daniel Luzzatto	941200511	daniel.luz@campus.technion.ac.il	28/10/2024

Supervisor	Email
Assist. Prof. Christian Grussler	

Faculty of Mechanical Engineering
Technion - Israel Institute of Technology
Spring 2024

1 Abstract

This study delves into the control theory problem of the inverted pendulum. The methods used are PID and LQR controllers, while the setup is the LEGO Spike Prime platform. After a theoretical analysis of the control strategies and simulation, the system was assembled, and state estimation techniques were implemented: a complementary filter was designed for accurate angle estimation by fusing data from the gyroscope and accelerometer and motor encoders were used to estimate the position and linear velocity, enabling precise state feedback for the LQR. The output were tested to ensure correctness.
The results showed that the system could be effectively stabilized using the LQR method, which offered superior control and robustness in handling the system's nonlinearity and external disturbances. In contrast, the PID controller faced more significant challenges, illustrating the advantages of LQR in complex control scenarios.

Contents

2 Theoretical background

Control theory is the subject that takes a physical problem, models it mathematically and implements a way to control the system such that it follows some goal behaviour. A key aspect is stability which is the tendency of the setup to converge to some value.
The Lego Segway simulates the problem of a pendulum on a cart: the challenge is balancing the system around an unstable point, making constant adjustments and minding that small deviations can lead to instability. Two control methods, PID and LQR, were implemented due to their ability to provide stability and fine-tuned responses. PID control is widely used, while LQR offers optimal control by minimizing a cost function based on some constraints to get the optimal response.

2.1 PID Controller

The first method used for stabilizing the robot is the PID controller (Proportional-Integral-Derivative controller) and it is based on the error between the setpoint and the output of the system. It is made out of three components:

- Proportional. This term corrects the signal proportionally to the error $e(t)$. This term reduces the error but does not eliminate it completely, as the system might show an oscillatory behaviour.
$$P = K_p \cdot e(t)$$
- Integral. This terms sums up the error accumulated over time and serves to correct steady state error.
$$I = K_i \cdot \int_0^t e(\tau)\, d\tau$$
- Derivative. This term predicts the future error by finding $\dot{e}(t)$ and serves to dampen the oscillations of the system.
$$D = K_d \cdot \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

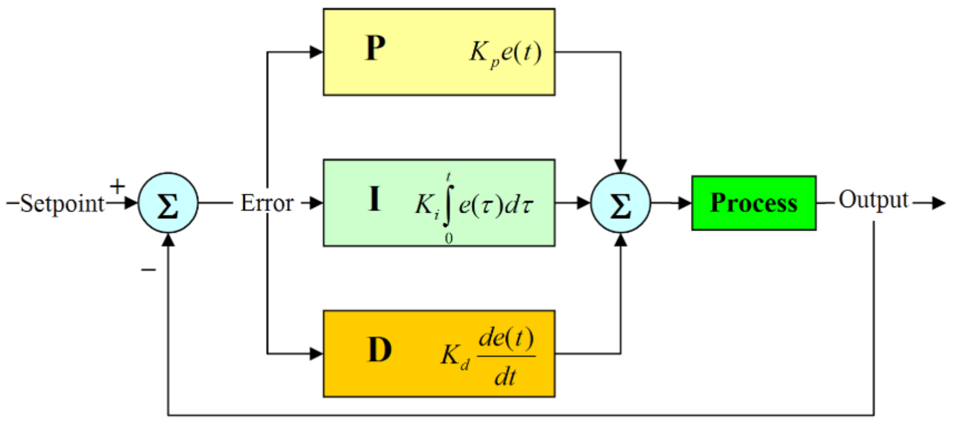


Figure 1: PID controller scheme

This controller is used in several applications, such as temperature control, speed control, and even balancing robots but has limitations systems with fast dynamics and significant noise.

2.2 LQR

The second method used is the LQR (Linear Quadratic Regulator). This regulator is a based on the state-space representation of the system and calculates the optimal gain matrix K which minimizes the cost function J balancing system performance and control effort:

$$J = \int_0^\infty x^T Q x + u^T R u \, dt$$

Where:

- x is the state vector.
- u is the control input.
- Q penalizes the deviations from the state variables, meaning that increasing this matrix, the system reacts more aggressively to deviations on the state variables.
- R penalizes control effort, meaning that increasing this matrix the system minimizes the energy usage.

Therefore the optimal control law is:

$$u = -Kx$$

More specifically, the gain matrix is found by solving the Riccati equation:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$

Where matrices A and B come from the state space representation, Q and R are chosen as penalties and P is the solution. Matrix K is found as follows:

$$K = R^{-1}B^T P$$

This method stabilizes the system around the equilibrium point for the given choice of the penalties and minimizing the cost.

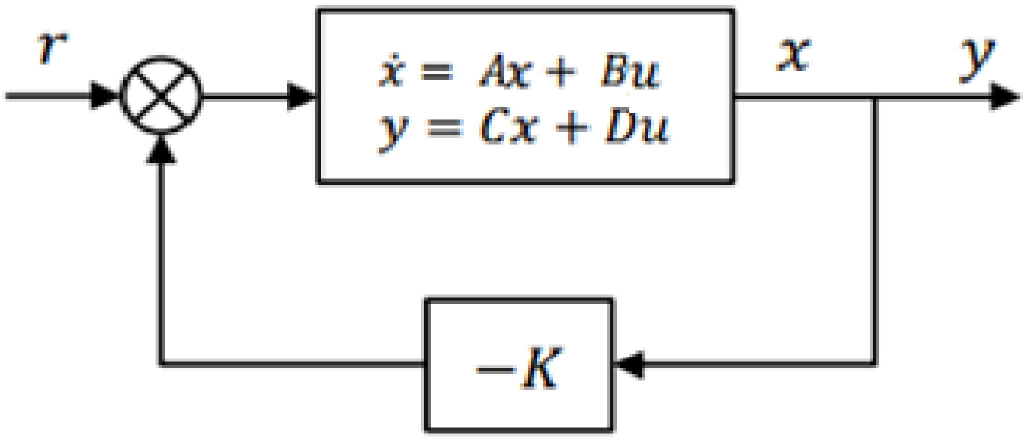


Figure 2: LQR scheme

2.3 Derivation of the equation of motion

In order to derive the equations of motion, the system was considered as a pendulum attached to a cart, as shown in the following picture.

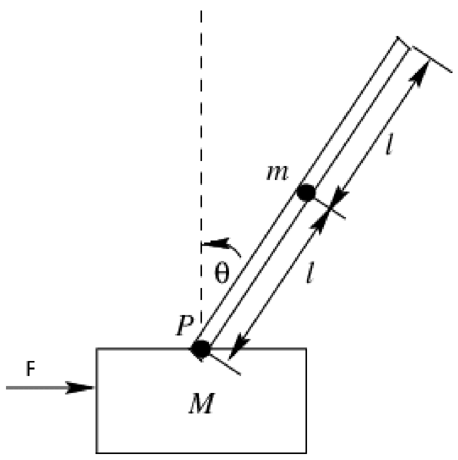
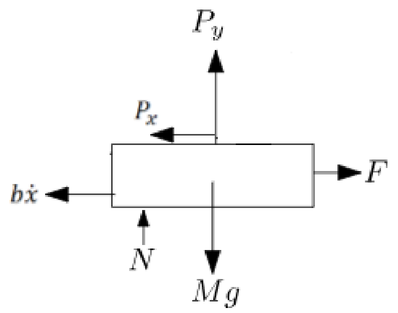


Figure 3: Pendulum on a cart

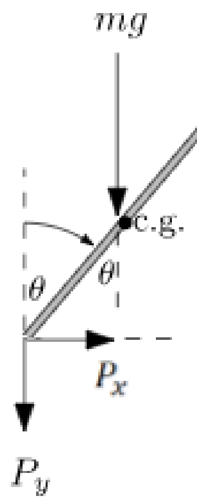
The following quantities describe the system:

- M: mass of the cart.
- m: mass of the pendulum.
- l: half length of the pendulum and location of the center of mass of the pendulum.
- I: moment of inertia of the pendulum.
- g: gravitational acceleration.
- x: position of the cart.
- θ: angle between the pendulum and the horizontal direction.

Let us draw the free body diagram of the cart and the pendulum:



(a) Free body diagram of the cart.



(b) Free body diagram of the pendulum.

The following forces act on the system:

- Gravitational force acting on the pendulum mg. Note that the one acting on the cart is balanced by the reaction force from the horizontal surface N.
- Friction force between wheels and the horizontal surface, $f = -b\dot{x}$.
- Reaction force between the pendulum and the cart, the horizontal component is denoted as P_x and the vertical component as P_y.
- Force applied horizontally to the cart, F.

The sum of the forces on the cart along the x axis yields:

$$\Sigma F_x = F - b\dot{x} - P_x = M\ddot{x} \tag{1}$$

We get the value of N using Newton's second law on the pendulum along the x axis:

$$P_x = ma$$

The acceleration a is found deriving twice the location of the center of mass in respect to time:

$$r_{cm} = \begin{bmatrix} x + l\sin\theta \\ l\cos\theta \end{bmatrix} \qquad v_{cm} = \begin{bmatrix} \dot{x} + l\dot{\theta}\cos\theta \\ -l\dot{\theta}\sin\theta \end{bmatrix} \qquad a_{cm} = \begin{bmatrix} \ddot{x} - l\dot{\theta}^2\sin\theta + l\ddot{\theta}\cos\theta \\ -l\dot{\theta}^2\cos\theta - l\ddot{\theta}\sin\theta \end{bmatrix}$$

It follows that:

$$P_x = m(\ddot{x} - l\dot{\theta}^2\sin\theta + l\ddot{\theta}\cos\theta) \tag{2}$$

Plugging into (1):

$$F - b\dot{x} - m(\ddot{x} - l\dot{\theta}^2\sin\theta + l\ddot{\theta}\cos\theta) = M\ddot{x}$$

$$\boxed{(M+m)\ddot{x} + b\dot{x} - ml\ddot{\theta}\cos\theta + ml\dot{\theta}^2\sin\theta = F} \tag{3}$$

The second equation is found by calculating the sum of moments around the center of mass of the pendulum and setting it equal to $-I\ddot{\theta}$ since θ was chosen to increase in the clockwise direction:

$$P_y l \sin\theta + P_x l \cos\theta = -I\ddot{\theta} \tag{4}$$

Applying Newton's second law to the pendulum along the y axis, we find P_y:

$$-P_y - mg = m(-l\dot{\theta}^2\cos\theta - l\ddot{\theta}\sin\theta)$$

$$P_y = m(l\dot{\theta}^2\cos\theta + l\ddot{\theta}\sin\theta) - mg \tag{5}$$

Plugging (2) and (5) into (4), we get:

$$(m(l\dot{\theta}^2\cos\theta + l\ddot{\theta}\sin\theta) - mg)l\sin\theta + m(\ddot{x} - l\dot{\theta}^2\sin\theta + l\ddot{\theta}\cos\theta)l\cos\theta = -I\ddot{\theta}$$

$$ml^2\dot{\theta}^2\cos\theta\sin\theta + ml^2\ddot{\theta}\sin^2\theta - mlg\sin\theta + m\ddot{x}l\cos\theta - ml^2\dot{\theta}^2\sin\theta\cos\theta + ml^2\ddot{\theta}\cos^2\theta = -I\ddot{\theta}$$

$$ml^2\ddot{\theta} - mlg\sin\theta + m\ddot{x}l\cos\theta = -I\ddot{\theta}$$

$$\boxed{(I + ml^2)\ddot{\theta} - ml\cos\theta\ddot{x} - mgl\sin\theta = 0} \tag{6}$$

Now we linearize the equations of motion around 0, using the small angle approximation (meaning $\sin\theta = \theta$ and $\cos\theta = 1$). This approximation is correct since the robot is supposed to oscillate between $-15°$ and $15°$ ($15° = 0.2618$ [rad]$\rightarrow$ err$= \frac{\sin(0.2618)-0.2618}{0.2618} \cdot 100 = 1.1384\%$).

$$\begin{cases} (M+m)\ddot{x} + b\dot{x} - ml\ddot{\theta} = F \\ (I+ml^2)\ddot{\theta} - ml\ddot{x} - mgl\theta = 0 \end{cases} \tag{7}$$

2.3.1 Transfer Function

Using the Laplace transform and its properties, it is possible to find a transfer function from the input force F to the angle θ. Applying the Laplace transform to (7) considering zero initial conditions:

$$\begin{cases} (M+m)X(s)s^2 + bX(s)s - ml\Theta(s)s^2 = F(s) \\ (I+ml^2)\Theta(s)s^2 - mlX(s)s^2 - mgl\Theta(s) = 0 \end{cases}$$

From the second equation, we solve for X:

$$X(s) = \frac{(I+ml^2)s^2 - mgl}{mls^2}\Theta(s) \tag{8}$$

Plug in $X(s)$ from (8) into the first equation:

$$(M+m)\frac{(I+ml^2)s^2 - mgl}{mls^2}\Theta(s)s^2 + b\frac{(I+ml^2)s^2 - mgl}{mls^2}\Theta(s)s - ml\Theta(s)s^2 = F(s) \tag{9}$$

After factoring out $\Theta(s)$, simplifying and normalizing the equation, we get the following transfer function from $F(s)$ to $\Theta(s)$:

$$\frac{\Theta(s)}{F(s)} = \frac{mls^2}{s^4 + \frac{b(I+ml^2)}{(M+m)(I+ml^2)-(ml)^2}s^3 - \frac{(M+m)mgl}{(M+m)(I+ml^2)-(ml)^2}s^2 - \frac{bmgl}{(M+m)(I+ml^2)-(ml)^2}s}$$

since there is a pole at the origin as well as a zero, we can cancel them out:

$$\boxed{\frac{\Theta(s)}{F(s)} = \frac{mls}{s^3 + \frac{b(I+ml^2)}{(M+m)(I+ml^2)-(ml)^2}s^2 - \frac{(M+m)mgl}{(M+m)(I+ml^2)-(ml)^2}s - \frac{bmgl}{(M+m)(I+ml^2)-(ml)^2}}}$$

2.3.2 State-Space Representation

From the equations of motion (7), we can find the state space representation of the system:

$$\begin{bmatrix} M+m & -ml \\ -ml & I+ml^2 \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} F-b\dot{x} \\ mgl\theta \end{bmatrix}$$

$$\begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \frac{1}{(M+m)(I+ml^2)-ml^2} \begin{bmatrix} I+ml^2 & ml \\ ml & M+m \end{bmatrix} \begin{bmatrix} F-b\dot{x} \\ mgl\theta \end{bmatrix}$$

$$\begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \frac{1}{I(M+m)+Mml^2} \begin{bmatrix} (I+ml^2)(F-b\dot{x})+m^2l^2g\theta \\ ml(F-b\dot{x})+(M+m)mgl\theta \end{bmatrix}$$

The state vector is $\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$, meaning the states are position, linear velocity, angle from the verical direction and angular velocity.
F represents the input, therefore will be written as u. The state space representation is:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-(I+ml^2)b}{I(M+m)+Mml^2} & \frac{m^2l^2g}{I(M+m)+Mml^2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{-mlb}{I(M+m)+Mml^2} & \frac{(M+m)mgl}{I(M+m)+Mml^2} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{I+ml^2}{I(M+m)+Mml^2} \\ 0 \\ \frac{ml}{I(M+m)+Mml^2} \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u$$

From (7), setting the time derivatives to zero, we find the equilibrium points of the system. It is easy to see that we get:

$$\sin\theta = 0 \quad \rightarrow \quad \theta = \pi n \ \text{ where } \ n = 0, 1$$

Since the solution are repeating, just the first two are considered. The linear velocity $\dot{x}$ and the angular velocity $\dot{\theta}$ are at equilibrium around 0 as well while the position can attain any value.

3 Simulations

Before testing the actual setup, simulations were made, for both the PID controller and the LQR.

3.1 PID controller

The controller was simulated on Simulink using the following:

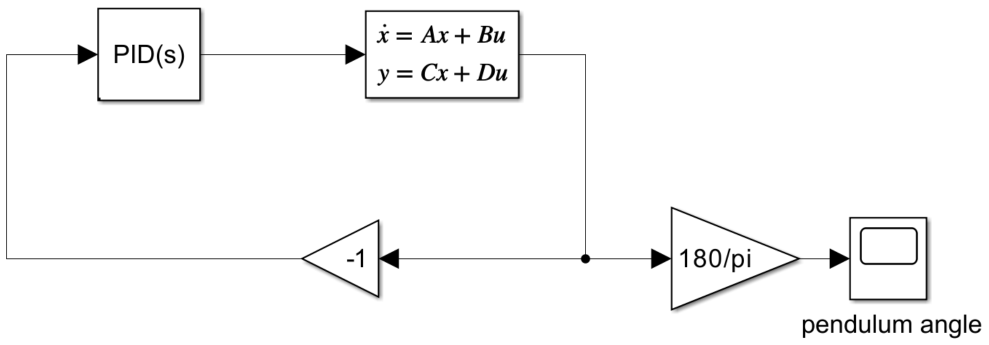


Figure 5: Simulink PID schematic

The matrices of the state space representation were A and B from section 2.3.2 while $C = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$ and $D = 0$ since we are controlling just the pendulum angle. The initial conditions were $x_0 = \begin{bmatrix} 0 & 0 & 2 & 0 \end{bmatrix}$, which is similar to what can be done with the real setup by holding it upright. The following PID parameters were chosen:

$$K_p = 150 \quad K_i = 10 \quad K_d = 200$$

The main strategy choosing them was starting with some proportional gain and derivative constant to make sure that the oscillations are dampened. Finally the integral constant serves to correct to some steady state error which comes up with the derivative part of the controller. Once the plot was close to the desired one, different combinations were computed (trial and error) to get the best response. Here is the result obtained from the simulation:

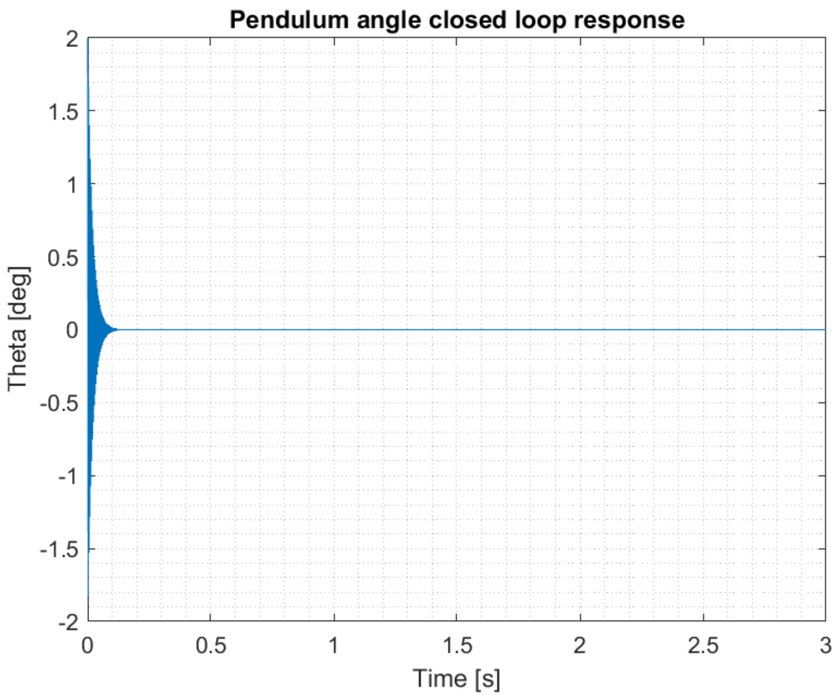


Figure 6: Pendulum angle time response

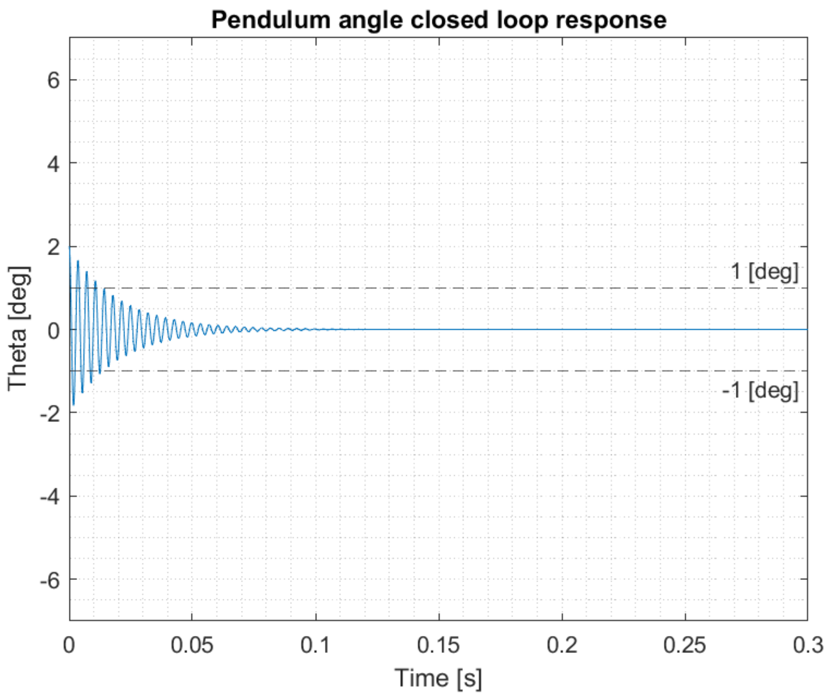


Figure 7: Zoom in of the pendulum time response

The pendulum angle response shows some small oscillations which decay within 0.1 seconds. The maximum angle reaches about $\pm 1.8°$ before settling around zero. Moreover, from figure 7 we notice that the oscillations are smaller than ± 1 degree after less than 0.02 seconds from the start of the simulation. The oscillations amplitude is reduced over time and the system stabilizes quickly, meaning that the controller is balancing well the pendulum and minimizing the oscillations caused by the initial

disturbance.
The code for this section is available at section 10.1

3.2 LQR

Next, the response to the LQR was simulated. Using the state-space matrices from section 2.3.2, after checking that the system is controllable and observable (condition for developing an LQR) the following weighting penalties matrices were chosen:

$$Q = 10^6 \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R = 0.01$$

The choice puts more emphasis on the state variables rather than on the control effort, as the magnitude of the diagonal entries of Q is much larger than R. Using the Matlab built-in function *lqr*, the gain matrix K was found:

$$K = 10^4 \cdot \begin{bmatrix} -1.0000 & -1.4658 & 5.6345 & 1.0935 \end{bmatrix}$$

Here is the response of the system to the same initial condition as in section 3.1, $x_0 = \begin{bmatrix} 0 & 0 & 2 & 0 \end{bmatrix}$:

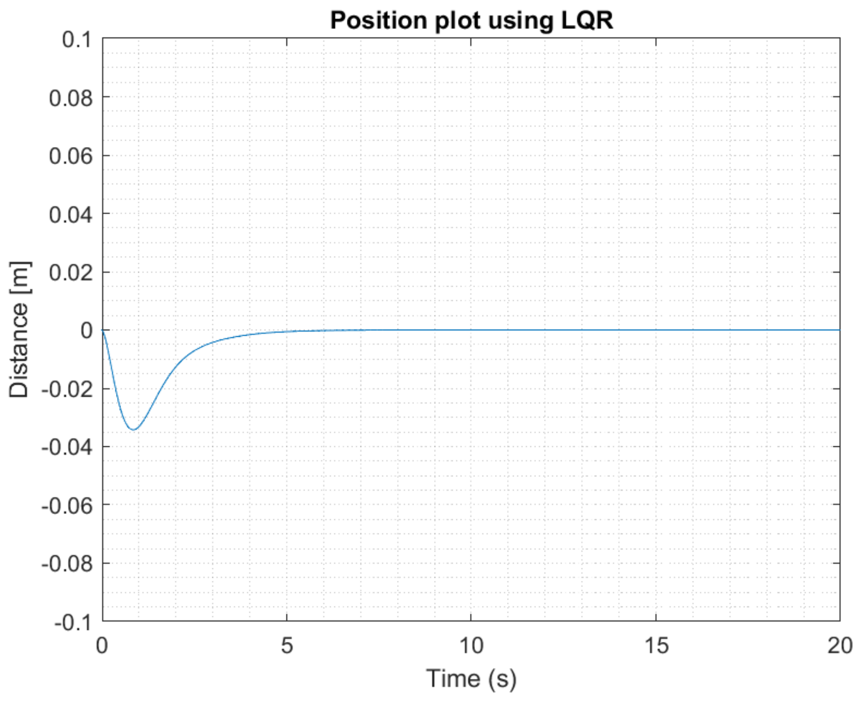


Figure 8: Position.

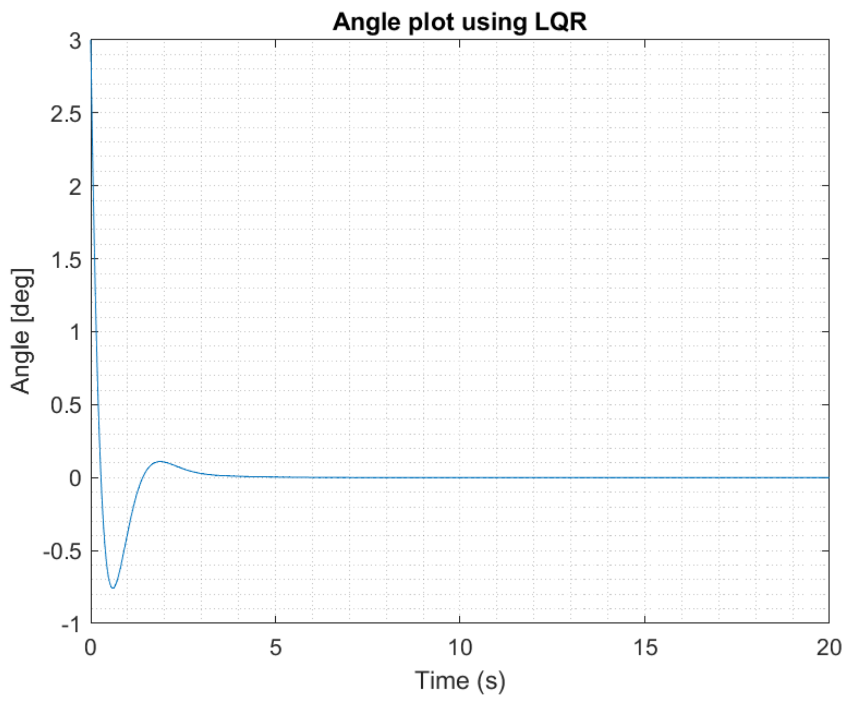


Figure 9: Angle.

As we can see from the above plots, position and angle converge quickly around the equilibrium point found in section 2.3.2 with little overshoot. Therefore, the LQR is stabilizing effectively the system with smooth responses, meaning Q and R were chosen accurately.
Comparing figure 9 to figures 6 and 7 from section 3.1, we notice a significant improvement, as the plots don't oscillates as much, even though it takes longer to converge to the steady state value.
The code for this section is available at section 10.1.

4 Physical setup

The physical setup used for the experiment is the LEGO set education SPIKE Prime. The robot was assembled in such a way to guarantee robustness, as outline below.

4.1 Components and Assembly

In the following figure are shown the components used for building the setup:

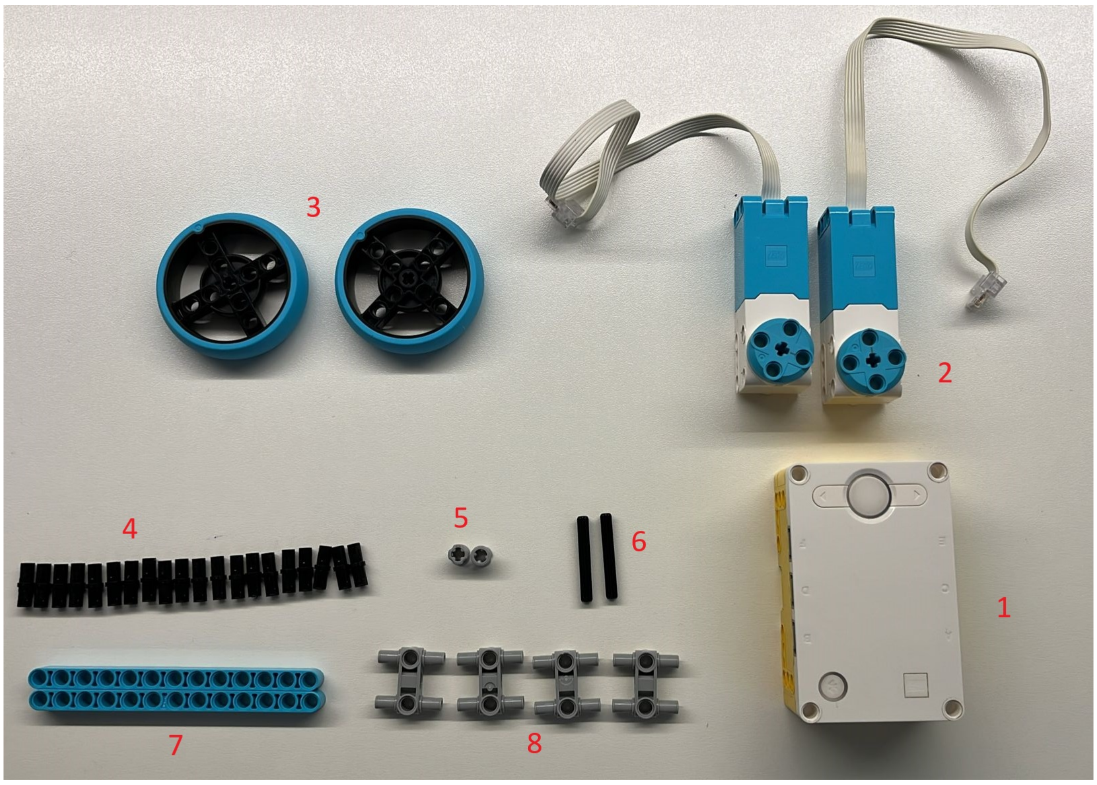


Figure 10: Components which make up the setup.

Number	Component	Quantity	Mass [g]
1	Hub and Battery	1	163
2	Medium Motor	2	50
3	Wheel ⌀56	2	15
4	Connector peg w/friction	20	< 1
5	Bush for cross axle	2	< 1
6	Cross axle 4m	2	1
7	Technic 13m beam	2	3
8	Beam 3 m w/4 snaps	4	1

Table 1: Components used for building the model. Check Figure 10 for the numbers. Note that the masses are used in section 4.1.1

The first step for building the segway is placing 4 *Connector peg w/friction* in each of the wheels, as shown in the following figure:

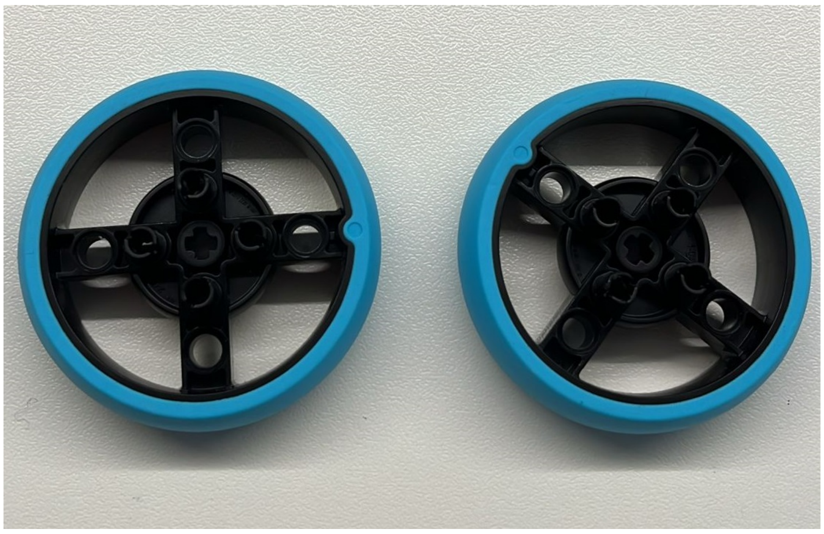


Figure 11: Wheel assembly.

Next, the wheels are assembled to the *Medium Motors* using the *Cross axle 4m* and secured using the *Bush for cross axle*:

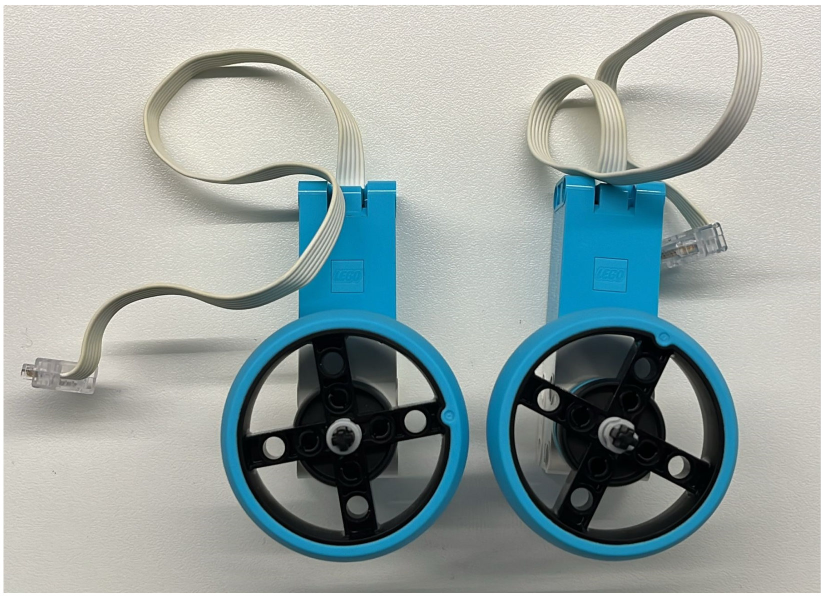


Figure 12: Motor-Wheels assembly.

At this stage, some mechanical play (backlash) was observed between the motors and the wheels, which results in a delay in the robot's response to control inputs, as the wheels could move even when the motors were not activated.
Afterwards, the Motor-Wheels assembly is attached to the *Hub and Battery* using 4 *Beam 3 m w/4 snaps* and 4 *Connector peg w/friction*. The wires connect the motors to the hub:

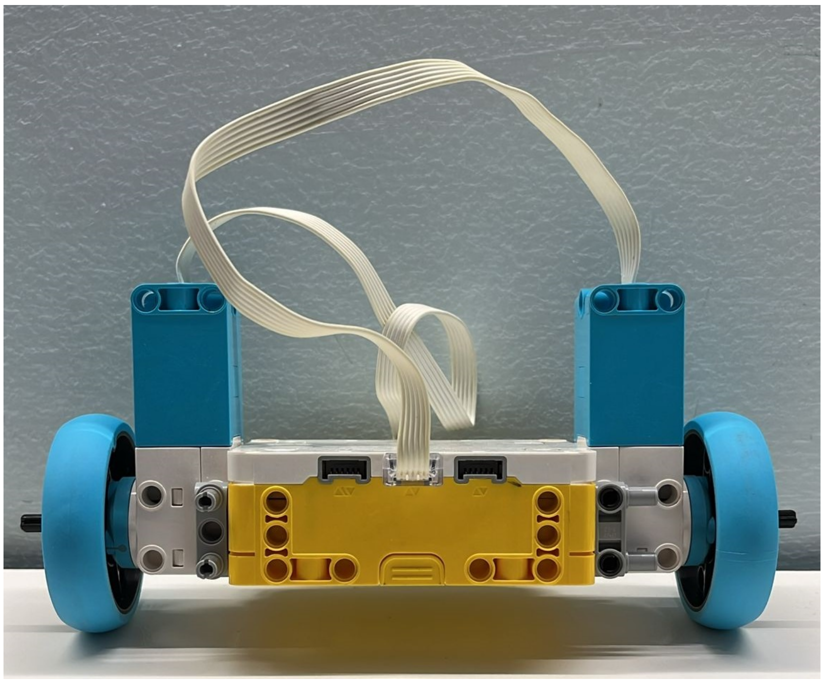


Figure 13: Motor-Wheels-Hub assembly.

As a last step, two *Technic 13m beam* connect the two *Medium Motors* to ensure proper positioning and straightness of the motors:

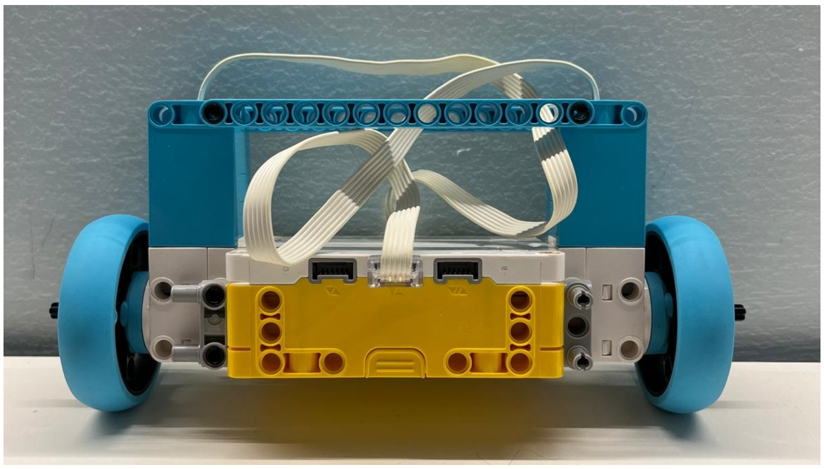


Figure 14: Full assembly.

The Lego segway is connected to the computer using bluetooth, through the SPIKE interface. This is useful as connecting the setup to the computer through a USB cable adds a tension force from the hub to the robot.

4.1.1 Moment of Inertia

In this section the moment of inertia is calculated based on the physical setup described in section 4.1. The result is used for the state space representation described in section 2.3.2. The calculation include Steiner's theorem (parallel axis theorem), as the rotation axis does not pass through the center of mass of the components.
The setup was divided in the components which make up the pendulum and the cart, as shown

in 2.3.2. The pendulum include those parts which rotate around the axis of rotation. Therefore, the wheels, the Cross axle 4m and the Bush for cross axle are considered part of the cart, while the remaining components are classified as pendulum. Additionally the Connector peg w/friction moment of inertia was neglected, as it was considered massless. The moment of inertia of the Beam 3m w/4 snaps was also neglected since their mass is low in respect to the rest and they are located at the axis of rotation, therefore there is no component from Steiner's theorem.
The system is approximated as 5 parallelepipeds: one making up the battery, two for the motors and two for the Technic 13m beam. The distance from each component's center of mass to the axis of rotation (axis of the wheels) is measured. The dimensions of the components perpendicular to the axis of rotation are denoted as h (height) and w (width), while the distance of the component center of mass to the axis of rotation as d:

$$\begin{aligned}
I_{battery} &= \frac{1}{12} m_{battery}(h_{battery}^2 + w_{battery}^2) + m_{battery} d_{battery}^2 \\
I_{motor} &= \frac{1}{12} m_{motor}(h_{motor}^2 + w_{motor}^2) + m_{motor} d_{motory}^2 \\
I_{beam} &= \frac{1}{12} m_{beam}(h_{beam}^2 + w_{beam}^2) + m_{beam} d_{beam}^2
\end{aligned}$$

The total moment of inertia is calculated as follows:

$$I = I_{battery} + 2I_{motor} + 2I_{beam}$$

The code for the calculation is shown in section 10.1.

4.2 Sensor Implementation and State Estimation

In order to control the Lego Segway, real time values for position, linear velocity, angle and angular velocity are required. They are obtained from the gyroscope, the accelerometer and the motor encoders. Gyroscope and accelerometer measure the angular velocity of the robot, the tilt angles respectively, while the motor encoders are used to measure the position and linear velocity of the robot. The code for this section is available at section 10.3 and 10.4.

4.2.1 Gyroscope

The function `get_gyro_rates()` retrieves the angular velocity along the x, y, and z axes. Since the angular velocity is given in decidegrees, the output must be divided by 10:

$$\omega_x, \omega_y, \omega_z = \frac{\text{angular velocity readings}}{10}$$

4.2.2 Accelerometer

The function `get_accel_angles()` calculates the tilt angles by measuring acceleration along the x, y, and z axes. The angle are computed using trigonometric relationships:

$$\text{roll} = \text{atan2}\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right), \quad \text{pitch} = \text{atan2}\left(\frac{-a_z}{a_x}\right)$$

4.2.3 Complementary filter

Relying on a single reading for the angle increases the noise and makes the measurement unreliable. Therefore, a complementary filter was used, as some possible errors from one of the sensors is compensated from the other. Here is the formula of the filter:

$$\theta = \alpha \cdot (\theta_{\text{prev}} + \omega_x \cdot \Delta t) + (1 - \alpha) \cdot \theta_{\text{accel}}$$

where $\alpha = \frac{\tau}{\tau + \Delta t}$ (τ is a tuning parameter), ω_x is the acceleration along the x axis, θ_{prev} is the tilt angle from the previous iteration and θ_{accel} is the tilt angle from the accelerometer.

4.2.4 Motor Encoders

The motor encoders are used to measure the position and linear velocity of the robot:

- Position: The motor encoders provide the relative position in degrees, which is then converted to meters using the wheel diameter:

  $$\text{position} = \frac{\text{relative\_position} \cdot \pi \cdot d}{360}$$

  where d is the diameter of the wheels.

- Linear Velocity: The built-in function `motor_velocity` gives a value in [decidegrees/s], therefore the conversion to [m/s] is the following:

  $$\text{linear velocity} = \frac{\text{motor velocity} \cdot \pi \cdot d}{36}$$

4.3 Testing Sensor Accuracy

To ensure that the sensor readings are accurate, some tests were conducted on the gyroscope, accelerometer and complementary filter.
The gyroscope is located in the *Hub and battery* part (check section 4.1). The Hub was positioned on a flat surface and the angular velocity was measured, checking if there is any drift over time.

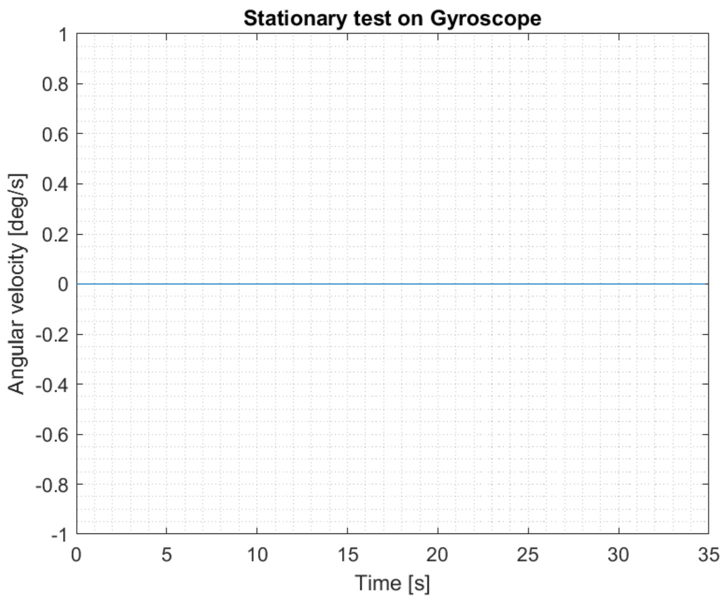


Figure 15: Stationary test for the gyroscope.

Since the plot is always at zero, the gyroscope doesn't show drift over time. Note that the plot shows only the angular velocity in the x direction as it's the only one used in stabilizing the robot; the other components are not shown for conciseness.

A similar test was conducted on the output of the accelerometer and of the complementary filter. Here are the results:

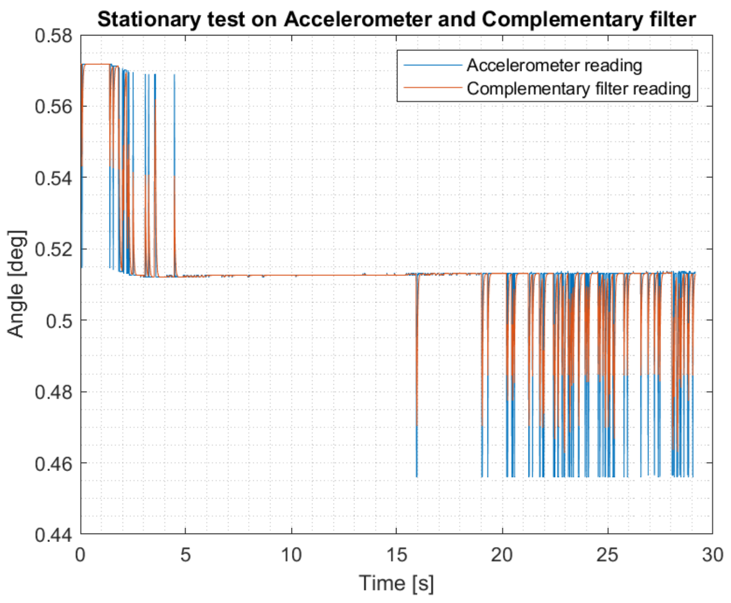


Figure 16: Stationary test for the accelerometer and complementary filter.

The difference between the maximal and minimal value obtained from the two measurments is low, being 0.1157 [deg] for the accelerometer reading and 0.1090 [deg]. Moreover, the root mean squared error between the two plots is minimal, being 0.0072 [deg]. Therefore, the test was successful. Another final test was performed, moving the robot and checking that the reading corresponds:

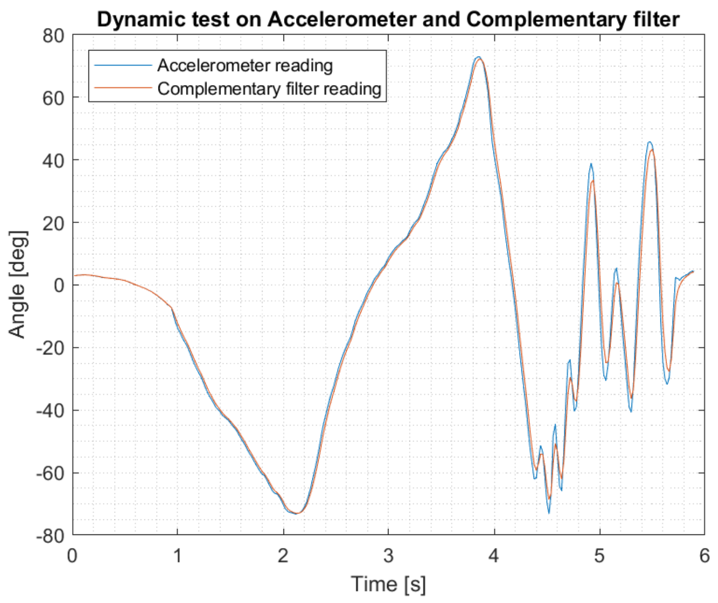


Figure 17: Dynamic test for the accelerometer and complementary filter.

Once again, the two readings match and the angle displayed matches with the orientation of the robot at that specifiic time frame. The root mean squared error being 3.5193 [deg]. Therefore, the angle measurements from the accelerometer and the complementary filter match.
The code for this section is available at section 10.2.

5 Results

5.1 PID Controller

The main goal was to control the robot's angle and maintain it within a range of $\pm 15°$ in respect to the vertical. The system's response and the choice of PID constants were influenced by the hardware characteristics of the motors and the dynamics of the robot. The code for this section is available at section 10.3.

5.1.1 Tuning strategy

Here is the tuning strategy used for the the PID controller:

1. The maximal angular velocity for the medium motors is $\pm 1110 \left[\frac{\text{deg}}{s}\right]$ and is the maximum control input to the system.

2. The first constant to tune is the proportional gain, leaving the other constants temporarily at 0. Consider the relation $u = k_p \cdot e$ (where u is the input to the system and e the error) and that the robot is supposed to stabilize within a range of $\pm 15°$, some value for the proportional constant was found:
$$K_p = \frac{u_{max}}{e} = \frac{1110}{15} = 74$$
However, considering the need to account for the derivative term, I opted for a lower K_p value, in the range of 30 to 35.

3. From the simulation conducted in section 3.1, the derivative component is supposed to be higher (but in the same order of magnitude) than the proportional gain.Therefore, I chose a derivative constant in the same range, around 35 to 40.

4. Since the integral term accumulates error over time and the robot is expected to oscillate, the integral component was kept small. I selected an integral constant of approximately 2.

5. As a last step, several combinations of numbers were tried in the range described.

5.1.2 Stabilization

The PID controller did not manage in stabilizing the robot even after tuning. Here are some of the possible reasons:

1. Friction. The robot stabilized when placed on a blanket (surface which provides more friction in respect to the floor and therefore, it dampened the oscillations) .during some test. Therefore the stability of the system might be sensitive to surface friction and disturbances.

2. Limitations of PID for this application. One of the key problems of using a PID controller for this application is the robot's rapid dynamics. The controller struggles to react fast enough to the changes in angle and velocity. Even after tuning, the robot tended to overcorrect or respond too weakly.

5.2 LQR

The experiment was performed using the gain matrix described in section 3.2 on the physical hardware described in section 4. Following are are the plots for the position from the test conducted on the robot:

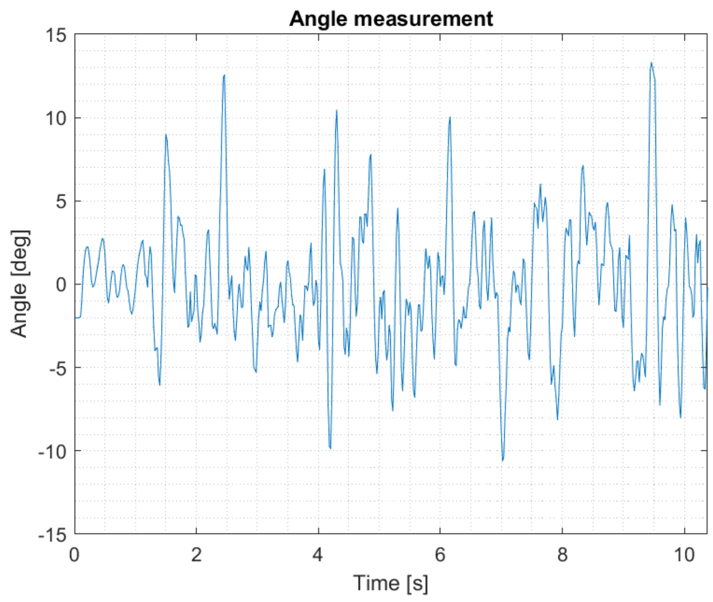


Figure 18: Angle plot.

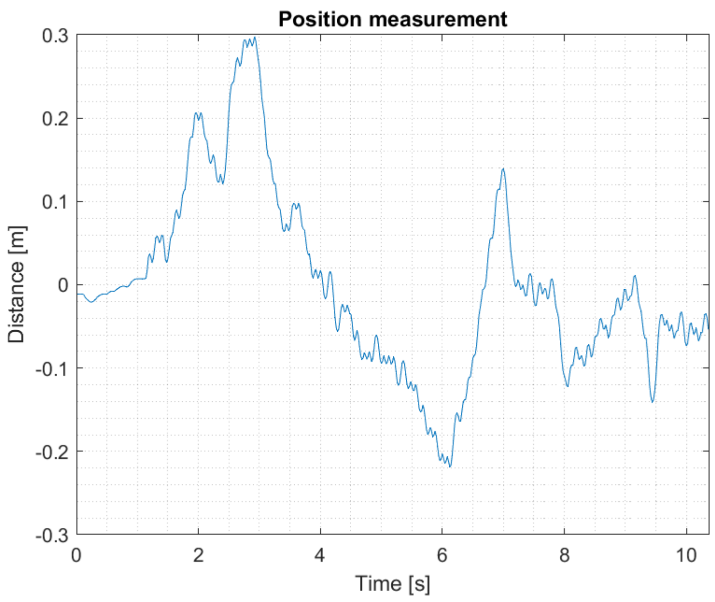


Figure 19: Position plot.

As we can see from Figure 18, the angle oscillates around the equilibrium point (which is 0). Most of the oscillations are within $\pm 10°$, which is an acceptable range for stabilizing the robot and for which the small angle approximation from section 2.3 is valid. The robot moving from one side to the other is an expected behaviour, as it tries to balance.
In Figure 19, we can see the postion plot of the robot. Even though the oscillations are larger than

the expected ones, the robot effectively reacts to a change in position and tends to balance the movement.
The code for this section is available at section 10.4.

6 Challenges Encountered

When testing the control strategies on the robot, some challenges arose. Following are the main ones.

1. Backlash was observed between the motors and the wheels, as the wheels would slightly rotated even if the motors were not moving. In order to reduce the effect of this issue, I added a small buffer in the control code to ignore minor changes in wheel rotation when the motors were not supposed to move. Despite the change, the issue wasn't solved for the PID controller and did not improve significantly the performance of the LQR.

2. During some tests, the robot stabilized when placed on a blanket, which provided more friction than a hard floor, helping reducing oscillations. This issue was noticed especially for the PID controller, which doesn't take into account the friction of the surface. The LQR does take into account of the friction as the gain matrix K is found from the state space representation which includes b, which is the friction coefficient, as shown in section 2.3. In order to get the most consistent behaviour, the robot was tested in several different surfaces and adjusted the PID constants accordingly.

3. The SPIKE python coding environment has some issues as it could not handle real-time data logging and plotting well, which affected the robot's performance during testing. The robot struggled to process control commands quickly while sending data for plotting, creating to delays in response which reduced instability. In order to overcome this issue, I tried reducing the data logging frequency and it helped improve the performances of the robot.

7 Conclusions

7.1 Discussion of the Results

As expected, the LQR controller performed better than the PID controller, both in the simulation and in the actual experiment, balancing effectively the segway. In contrast, the PID controller struggled to cope with the rapid dynamics of the system and didn't stabilize the robot.
The simulation results for the LQR controller matched well with the experimental data, as the gain matrix from the simulation gave a good response worked for both, confirming the validity of the state-space model and the control parameters chosen for the LQR. On the other hand, for the PID controller, the simulation and test results were different: the simulation predicted stabilization after some oscillation, while the PID controller could not keep the system stable.
The LQR worked thanks to its optimal design, balancing system performance and control effort, while the PID controller either overcorrected or didn't correct fast enough to the system's dynamics.

7.2 Improvements and Future Work

While the setup was successfully stabilized using the LQR, there are some areas of improvement which could enhance the performance of the system.
The first is state estimation, as a Kalman filter could help getting better values for the state space variables and dealing with unexpected disturbances.
Dealing with surface conditions is another area of improvements: the robot's physical design can be changed, maybe by using rubberized wheels, which would improve the performances of the robot.
Lastly, solving the backlash issue which can be challenging due to the LEGO construction, but it'd be worth reducing this play by using tighter fittings or improved mounts.

8 Bibliography and References

[1] Karl Johan Astrom and Richard M. Murray. Feedback systems. Princeton University Press, Princeton, NJ, February 2021.

[2] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. Feedback control of dynamic systems. Pearson, Upper Saddle River, NJ, 8 edition, January 2018.

[3] https://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling

[4] https://www.12000.org/my_notes/cart_motion/report.htm

[5] https://le-www-live-s.legocdn.com/sc/media/files/support/spike-prime/le_spike_prime_set_element_overview_classroom_poster_18x24inch-a7ecd36fbf6d15fd4c7617f4cb882531.pdf

[6] https://spike.legoeducation.com/essential/help/lls-help-python#lls-help-python-spm-motor_pair-func-move

9 Appendix

9.1 Simulation Code

Here is the matlab code used for section 3.

```
close all;clc

L = 9*10^-2; %cm, full length pendulum
g = 9.81; %m/s^2
l = L/2; % half length of the pendulum
b = 0.01; % friction coefficient

diam = 0.056;
r = diam/2;
h_motor = 0.0715;
w_motor = 0.023;
h_battery = 0.0315;
w_battery = 0.056;
m_tot = 0.292;
m_wheel =0.015; % cart (wheels)
m_motor = 0.05;
m_battery = 0.163;
m_beam = 0.003;
h_beam = 0.007;
w_beam = 0.0075;
a = 0.009; %distance from the bottom part of the battery to the axis of rotation
d_motor = h_motor/2-a;
d_battery = h_battery/2-a;
d_beam = 0.0715-a;

inertia_motor = 1/12*m_motor*(h_motor^2+w_motor^2)+m_motor*d_motor^2;
inertia_battery = 1/12*m_battery*(h_battery^2+w_battery^2)+m_battery*d_battery^2;
inertia_beam = 1/12*m_beam*(h_beam^2+w_beam^2)+m_beam*d_beam^2;
I = inertia_battery+2*inertia_motor+ 2*inertia_beam;

M = 2*m_wheel; % cart
m = m_tot-2*m_wheel; % pendulum

% State Space Representation
A = [0, 1, 0, 0;
    0, -(I + m*l^2)*b / (I*(M+m) + M*m*l^2), (m^2 * g * l^2) / (I*(M+m) + M*m*l
    ^2), 0;
    0, 0, 0, 1;
    0, -(m*l*b) / (I*(M+m) + M*m*l^2), m*g*l*(M+m) / (I*(M+m) + M*m*l^2), 0];

B = [0;
    (I + m*l^2) / (I*(M+m) + M*m*l^2);
    0;
    m*l / (I*(M+m) + M*m*l^2)];

C = [1, 0, 0, 0;
    0, 0, 1, 0];

D = [0; 0];


%% PID Simulation
```

```
close all; clc
data = sim("segway_linearizaton_PID.slx");

theta = data.pend_angle.signals.values;
time = data.tout;
figure;
plot(time,theta)
grid minor
xlabel('Time [s]')
ylabel('Theta [deg]')
yline(1,'--','1 [deg]','LabelVerticalAlignment','top')
yline(-1,'--', '-1 [deg]','LabelVerticalAlignment','bottom')
title('Pendulum angle closed loop response')
xlim([0 0.3])
ylim([-7 7])
saveas(gcf, 'Simulation_PID_zoom.png')

figure;
plot(time,theta)
grid minor
xlabel('Time [s]')
ylabel('Theta [deg]')
title('Pendulum angle closed loop response')
xlim([0 3])
saveas(gcf, 'Simulation_PID.png')
%% LQR Simulation
clc; close all
sys = ss(A, B, C, D);
tf_sys = tf(sys);

contr = rank(ctrb(sys));
obs = rank(obsv(sys));
eigen_A = eig(A);
Q = 10^6*eye(4);
R = 0.01;
K = -lqr(sys,Q,R);

A_cl = A + B * K;
sys_cl = ss(A_cl, B, C, D);

t = linspace(0, 20, 20*40);
x0 = [0, 0, 3*pi/180, 0];  % [position, velocity, angle, angular velocity] IC

[y_out, t_out, x_out] = initial(sys_cl, x0, t);

figure;
plot(t_out, x_out(:, 1));
title('Position plot using LQR');
xlabel('Time (s)');
ylabel('Distance [m]');
grid minor;
ylim([-0.1 0.1])
saveas(gcf, 'pos_lqr_simulation.png')

figure;
plot(t_out, x_out(:, 2));
title('Linear velocity plot using LQR');
xlabel('Time (s)');
```

```matlab
ylabel('Linear Velocity [m/s]');
grid minor;
saveas(gcf, 'linvel_lqr_simulation.png')

figure;
plot(t_out, x_out(:, 3)*180/pi);
title('Angle plot using LQR');
xlabel('Time (s)');
ylabel('Angle [deg]');
grid minor;
saveas(gcf, 'angle_lqr_simulation.png')

figure;
plot(t_out, x_out(:, 4));
title('Angular velocity plot using LQR');
xlabel('Time (s)');
ylabel('Angular Velocity [deg/s]');
grid minor;
saveas(gcf, 'angvel_lqr_simulation.png')
```

9.2 Sensor Testing code

Here is the matlab code used for section 4.

```matlab
clc; close all
gyro_data = readtable('drift_gyroscope.csv');
gyro_time = gyro_data.timestamp;
gyro_drift = gyro_data.red;

comp_stat = readtable('accel_compl_stattest.csv');
comp_time = comp_stat.timestamp;
comp_accelstat= comp_stat.red;
comp_filtstat= comp_stat.orange;
avg_accelstat = mean(comp_accelstat);
avg_filtstat = mean(comp_filtstat);
maxdev_accelstat = max(comp_accelstat)-min(comp_accelstat);
maxdev_filtstat = max(comp_filtstat)-min(comp_filtstat);
RMSE_stat = sqrt(mean((comp_filtstat - comp_accelstat).^2));

dyn_test = readtable('dynamic_test.csv');
dyn_time = dyn_test.timestamp;
dyn_accel = dyn_test.red;
dyn_filt = dyn_test.orange;
RMSE_dyn = sqrt(mean((dyn_accel - dyn_filt).^2));

figure;
plot(gyro_time, gyro_drift)
grid minor;
xlabel('Time [s]')
ylabel('Angular velocity [deg/s]')
title('Stationary test on Gyroscope')
saveas(gcf, 'gyro_drift.png')

figure;
plot(comp_time, comp_accelstat)
hold on
plot(comp_time, comp_filtstat)
grid minor;
```

```
xlabel('Time [s]')
ylabel('Angle [deg]')
legend('Accelerometer reading', 'Complementary filter reading')
title('Stationary test on Accelerometer and Complementary filter')
saveas(gcf, 'accel_filt_stat.png')

figure;
plot(dyn_time, dyn_accel)
hold on
plot(dyn_time, dyn_filt)
grid minor;
xlabel('Time [s]')
ylabel('Angle [deg]')
legend('Accelerometer reading', 'Complementary filter reading', 'Location','
    northwest')
title('Dynamic test on Accelerometer and Complementary filter')
saveas(gcf, 'dyn_test.png')
```

9.3 PID controller

Here is the python code used for section 5.1 to control the physical setup:

```
import runloop, motor_pair
from hub import port, motion_sensor
import time
import math

def get_gyro_rates():
    omega_x, omega_y, omega_z = motion_sensor.angular_velocity(False)
    return omega_x/10, omega_y/10, omega_z/10

def get_accel_angles():
    accel_x, accel_y, accel_z = motion_sensor.acceleration(False)
    roll_angle = math.atan2(accel_y, math.sqrt(accel_x**2 + accel_z**2)) * 180 /
    math.pi# around z-axis
    pitch_angle = math.atan2(-accel_z, accel_x) * 180 / math.pi# around y-axis
    return pitch_angle+90, roll_angle


async def main():
    time.sleep(1)

    target_roll = 0
    power = 1
    k_p = 35
    k_i =2
    k_d =40

    error = 0
    integral = 0
    derivative = 0
    previous_error = 0

    tau = 0.5
    fs = 70
    dt = 1 / fs
    a = tau / (tau + dt)

    pitch_angle_accel, roll_angle_accel = get_accel_angles()
    angle = roll_angle_accel

    motor_pair.unpair(motor_pair.PAIR_1)
    motor_pair.pair(motor_pair.PAIR_1, port.C, port.D)

    while True:
        gyro_rate_x, gyro_rate_y, gyro_rate_z = get_gyro_rates()
        pitch_angle_accel, roll_angle_accel = get_accel_angles()

        angle = a * (angle + gyro_rate_x * dt) + (1 - a) * roll_angle_accel
        error = -(target_roll - angle) #[deg]
        integral += error * dt #[deg*s]
        derivative = (error - previous_error) / dt #[deg/s]
        result_power = k_p * error + k_i * integral + k_d * derivative
        movement_speed = int(result_power * power)

        motor_pair.move(motor_pair.PAIR_1, 0, velocity=movement_speed) # max speed = +-1110 deg/s
        previous_error = error

        await runloop.sleep_ms(int(dt * 1000))

runloop.run(main())
```

9.4 LQR

Here is the python code used for section 5.2 to control the physical setup:

```
from app import linegraph
import runloop, motor_pair, motor
from hub import port, motion_sensor
import time
import math

diam = 0.056

def get_gyro_rates():
    omega_x, omega_y, omega_z = motion_sensor.angular_velocity(False)
    return omega_x / 10, omega_y / 10, omega_z / 10

def get_accel_angles():
    accel_x, accel_y, accel_z = motion_sensor.acceleration(False)
    roll_angle = math.atan2(accel_y, math.sqrt(accel_x**2 + accel_z**2)) * 180 / math.pi
    pitch_angle = math.atan2(-accel_z, accel_x) * 180 / math.pi
    return pitch_angle + 90, roll_angle

async def main():
    time.sleep(1)
    K = [-1.0000, -1.4658, 5.6345, 1.0935]
    for i in range(len(K)):
        K[i] = pow(10,4)*K[i]
    print(K)
    fs = 50
    dt = 1 / fs
    motor_pair.unpair(motor_pair.PAIR_1)
    motor_pair.pair(motor_pair.PAIR_1, port.C, port.D)
    x = [0, 0, 0, 0] # State vector [position, linear velocity, angle, angular
velocity]
    pitch_angle_accel, roll_angle_accel = get_accel_angles()
    angle = roll_angle_accel
    a = 0.5
    u = 0
    current_time = 0
    pos_init = motor.relative_position(2) * diam * dt / 2
    linegraph.clear_all()

    while True:
        gyro_rate_x, gyro_rate_y, gyro_rate_z = get_gyro_rates()
        pitch_angle, roll_angle = get_accel_angles()
        # Complementary filter
        angle = a * (angle + gyro_rate_x * dt) + (1 - a) * roll_angle

        lin_vel = motor.velocity(2) * diam * math.pi / 36 # Convert velocity to
meters/sec
        pos = motor.relative_position(2) * diam * math.pi / 360 - pos_init

        x = [pos, lin_vel, angle, gyro_rate_x]
        u = -sum([K[i] * x[i] for i in range(4)]) #control law
        movement_speed = int(u)
        motor_pair.move(motor_pair.PAIR_1, 0, velocity=movement_speed)
        current_time +=dt
        linegraph.plot(9, current_time, angle)
        linegraph.plot(8, current_time, pos)
        linegraph.show(False)

        await runloop.sleep_ms(int(dt * 1000))

runloop.run(main())
```

Following is the matlab code used for handling the data and plotting the results from the LQR testing:

```
lqr_test = readtable('LQR_plotf.csv');
time_lqr = lqr_test.timestamp;
angle_lqr = lqr_test.red;
pos_lqr = lqr_test.orange;

figure;
plot(time_lqr, angle_lqr)
grid minor
xlabel('Time [s]')
ylabel('Angle [deg]')
title('Angle measurement')
xlim([0, time_lqr(end)])
saveas(gcf, 'lqr_angle.png')

figure;
plot(time_lqr, pos_lqr)
grid minor
xlabel('Time [s]')
ylabel('Distance [m]')
title('Position measurement')
xlim([0, time_lqr(end)])
saveas(gcf, 'lqr_pos.png')
```